DD2457 Program Semantics and Analysis

EXAMINATION PROBLEMS
WITH PARTIAL SOLUTIONS
21 May 2013

Dilian Gurov
KTH CSC
tel: 08-790 8198

Give solutions in English or Swedish, each problem beginning on a new sheet. Write your name on all sheets. The maximal number of points is given for each problem. The total number of points is 30. Up to two bonus points per section will be taken into account. The course book, the handouts, own notes taken in class, as well as reference material are admissible at the exam.

1 Level E

For passing level E you need 6 (out of 8) points from this section.

Recall the language extensions to **While** we considered in the second homework, with statements **read** x and **write** a for reading in a value from the keyboard and storing it in variable x, and for writing the value of expression a (in the current state) to the console, respectively. Also recall statement **thread** S **end** for writing multi–threaded programs. We noticed that extending the *structural operational semantics* of the language to deal with threads is not straightforward. As one possible solution we mentioned the following approach. We change the configurations to be of shape $\langle \mathcal{M}, s\rangle$, where $\mathcal{M}$ is a multi–set of statement sequences $\gamma \in \mathbf{Stm}^*$, representing the threads in the system. Then, the rule for **skip** can look like this:

$$[\mathsf{skip_{MT}}] \quad \langle (\mathbf{skip} : \gamma) + \mathcal{M}, s\rangle \Rightarrow \langle \gamma + \mathcal{M}, s\rangle$$

where we use the symbolic–sum notation for multi–sets (in which, for example, expression $2 \cdot a + 3 \cdot b$ denotes the multi–set with two occurrences of element a and three occurrences of b). By convention, let's ignore the empty sequence in this notation (corresponding to completed threads). The rules always inspect the head of one of the statement sequences in the multi–set, chosen non–deterministically. Initial configurations are again of shape $\langle S, s\rangle$, i.e. they have one single–element list as a first component, while final configurations are of shape s, i.e. all threads have completed.

1. Complete the suggested operational semantics (MT) in the style discussed above, for **While** extended with input, output and threads. Recall that input and output transitions are labelled with the corresponding side–effect $?z$ or $!z$. Give names to the rules as suggested by the above example rule. $\boxed{4p}$

   Hint: The intention is that sequential composition is split into the sub–statements, while **thread** S **end** spawns off a new thread for executing statement S. There should be axiom rules only.

   Solution: Here is a possible set of rules:

$$\begin{array}{rll}
[\mathsf{skip_{MT}}] & \langle (\mathbf{skip} : \gamma) + \mathcal{M}, s\rangle \Rightarrow \langle \gamma + \mathcal{M}, s\rangle & \\
[\mathsf{ass_{MT}}] & \langle (x := a : \gamma) + \mathcal{M}, s\rangle \Rightarrow \langle \gamma + \mathcal{M}, s[x \mapsto \mathcal{A}[\![a]\!](s)]\rangle & \\
[\mathsf{read_{MT}}] & \langle (\mathbf{read}\ x : \gamma) + \mathcal{M}, s\rangle \overset{?z}{\Longrightarrow} \langle \gamma + \mathcal{M}, s[x \mapsto z]\rangle & \\
[\mathsf{write_{MT}}] & \langle (\mathbf{write}\ a : \gamma) + \mathcal{M}, s\rangle \overset{!\mathcal{A}[\![a]\!](s)}{\Longrightarrow} \langle \gamma + \mathcal{M}, s\rangle & \\
[\mathsf{thread_{MT}}] & \langle (\mathbf{thread}\ S\ \mathbf{end} : \gamma) + \mathcal{M}, s\rangle \Rightarrow \langle S + \gamma + \mathcal{M}, s\rangle & \\
[\mathsf{comp_{MT}}] & \langle (S_1; S_2 : \gamma) + \mathcal{M}, s\rangle \Rightarrow \langle (S_1 : S_2 : \gamma) + \mathcal{M}, s\rangle & \\
[\mathsf{if^{tt}_{MT}}] & \langle (\mathbf{if}\ b\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2 : \gamma) + \mathcal{M}, s\rangle \Rightarrow \langle (S_1 : \gamma) + \mathcal{M}, s\rangle & \text{if } \mathcal{B}[\![b]\!](s) = \mathbf{tt} \\
[\mathsf{if^{ff}_{MT}}] & \langle (\mathbf{if}\ b\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2 : \gamma) + \mathcal{M}, s\rangle \Rightarrow \langle (S_2 : \gamma) + \mathcal{M}, s\rangle & \text{if } \mathcal{B}[\![b]\!](s) = \mathbf{ff} \\
[\mathsf{while^{tt}_{MT}}] & \langle (\mathbf{while}\ b\ \mathbf{do}\ S : \gamma) + \mathcal{M}, s\rangle \Rightarrow \langle (S : \mathbf{while}\ b\ \mathbf{do}\ S : \gamma) + \mathcal{M}, s\rangle & \text{if } \mathcal{B}[\![b]\!](s) = \mathbf{tt} \\
[\mathsf{while^{ff}_{MT}}] & \langle (\mathbf{while}\ b\ \mathbf{do}\ S : \gamma) + \mathcal{M}, s\rangle \Rightarrow \langle \gamma + \mathcal{M}, s\rangle & \text{if } \mathcal{B}[\![b]\!](s) = \mathbf{ff}
\end{array}$$

2. Use your semantics to explore the configuration space of the following program: $\boxed{\text{3p}}$

$$\textbf{while true do } (\textbf{read } x;\ \textbf{thread write } 1 - x \textbf{ end})$$

from a state s such that $s(x) = 0$, assuming only 0's and 1's are entered as input. Draw an informative subgraph of the configuration graph. Introduce shorthands for certain compound statements to make the graph easier to read.

3. Formulate at least two relevant properties of the input–output behaviour of the above program. $\boxed{\text{1p}}$

Solution: In every partial execution of the program:

(a) there are never more outputs than inputs; and

(b) every output value is equal to the negation of the latest input value.

---

2 Level C

For grade D you need to have passed level E and obtained 5 (out of 12) points from this section. For passing level C you need 8 points from this section.

In this section we will develop an *Interval Analysis* through abstract interpretation. Consider a possible extension of the **While** language with *arrays*. An array declaration of the form:

$$\textbf{array } x[10];$$

can then be understood as declaring 10 variables $x[0]$, $x[1]$, …, $x[9]$. Interval analysis is a generalization of the Constant Propagation Analysis discussed in class. If the analysis assigns the interval $[-5, 7]$ to variable x at a given point in a program, we can deduce that the values of x at this program point can only be within this range. Constant values could then be viewed as intervals of the form $[n, n]$. The analysis is mainly applied for finding *array–out–of–bounds* errors in programs, as for instance a statement in the program mentioning variable $x[j]$ in a state where j evaluates to an index outside of the declared interval.

The abstract domain of values can be defined as:

$$\mathbf{I} \overset{\mathsf{def}}{=} \{[i,j] \mid i,j \in \mathbf{Z} \cup \{-\infty, +\infty\}, i \le j\} \cup \{[\,]\}$$

where $[\,]$ denotes the empty interval.

1. Describe the lattice of abstract values you consider. In particular, how do you define lub's $\sqcup$ on intervals, and what are the atomic and the top and bottom elements of the lattice? $\boxed{\text{2p}}$

Solution: The atomic values are the intervals of shape $[i, i]$. The top element is the interval $[-\infty, +\infty]$, while the bottom element is $[\,]$. The lub of intervals $[i_1, j_1]$ and $[i_2, j_2]$ will then be the interval $[min(i_1, i_2), max(j_1, j_2)]$.

2. Define the abstraction function $\mathsf{abs}_Z : \mathbf{Z} \to \mathbf{I}$, the operations $+_I$, $-_I$ and $\star_I$ and relations $=_I$ and $\le_I$ on $\mathbf{I}$. Argue for your choice of definition in terms of safety and precision. The operations $\neg_I$ and $\wedge_I$ on $\mathbf{TT}$, and the semantic functions $\mathcal{IA}$ and $\mathcal{IB}$ can be defined as in the Detection of Signs analysis, and don't have to be explicitly defined here. $\boxed{\text{3p}}$

Solution: The abstraction function can be defined by $\mathsf{abs}_Z(z) \overset{\mathsf{def}}{=} [z, z]$. Addition $+_I$ on intervals is best defined by $[i_1, j_1] +_I [i_2, j_2] \overset{\mathsf{def}}{=} [i_1 + i_2, j_1 + j_2]$, since the latter interval is the narrowest (and thus most precise) interval containing all possible sums of an integer from the first interval with an integer from the second interval. The other operations can be defined analogously.

The relation $\leq_I$ on intervals can be defined by

$$[i_1, j_1] \leq_I [i_2, j_2] \overset{\mathsf{def}}{=} \begin{cases} \text{NONE} & \text{if at least one of the intervals is empty} \\ \text{TT} & \text{if } j_1 \leq i_2 \\ \text{FF} & \text{if } j_2 < i_1 \\ \text{ANY} & \text{otherwise} \end{cases}$$

The relation $=_I$ can be defined analogously.

3. Develop the abstract interpretation of statements in a semantic style of your choice, i.e., either through operational semantics rules, or by defining the denotational style semantic function $\mathcal{IS}$ (don't forget to define the semantic conditional $\mathsf{cond}_I(f, h_1, h_2)$ in that case). $\boxed{\text{3p}}$

   Solution: If one chooses the SOS semantics style, the rules are exactly the ones we had for the Detection of Signs analysis in SOS style; see homework assignment five.

4. Apply your interval analysis to the following program: $\boxed{\text{4p}}$

```
\\ array x[2];
y := 2;
i := 0;
while i ≤ y do
    \\ ...x[i]...;
    i := i + 1;
```

where the statements using the language extension are commented out. What can you deduce about possible optimizations and array–out–of–bounds errors? Explain how you use your abstract interpretation to perform the analysis. For instance, if your abstract interpretation is in SOS operational style, how does the analysis utilize configuration space exploration?

Solution: In SOS style, we start from the property state ps = TOP. For this particular program it doesn't matter much in what order we combine configuration space exploration with computing lub's for configurations with the same statement as first component. We can do this as in the second lab assignment, by first computing the configuration space (which will terminate) and then computing the lub's. Here is the only execution:

$$\begin{array}{ll}
 & \langle y := 2; i := 0; \textbf{while } i \leq y \textbf{ do } i := i+1, ps\rangle \\
\Rightarrow & \langle i := 0; \textbf{while } i \leq y \textbf{ do } i := i+1, ps[y \mapsto [2,2]]\rangle \\
\Rightarrow & \langle \textbf{while } i \leq y \textbf{ do } i := i+1, ps[y \mapsto [2,2], i \mapsto [0,0]]\rangle \\
\Rightarrow & \langle i := i+1; \textbf{while } i \leq y \textbf{ do } i := i+1, ps[y \mapsto [2,2], i \mapsto [0,0]]\rangle \\
\Rightarrow & \langle \textbf{while } i \leq y \textbf{ do } i := i+1, ps[y \mapsto [2,2], i \mapsto [1,1]]\rangle \\
\Rightarrow & \langle i := i+1; \textbf{while } i \leq y \textbf{ do } i := i+1, ps[y \mapsto [2,2], i \mapsto [1,1]]\rangle \\
\Rightarrow & \langle \textbf{while } i \leq y \textbf{ do } i := i+1, ps[y \mapsto [2,2], i \mapsto [2,2]]\rangle \\
\Rightarrow & \langle i := i+1; \textbf{while } i \leq y \textbf{ do } i := i+1, ps[y \mapsto [2,2], i \mapsto [2,2]]\rangle \\
\Rightarrow & \langle \textbf{while } i \leq y \textbf{ do } i := i+1, ps[y \mapsto [2,2], i \mapsto [3,3]]\rangle \\
\Rightarrow & ps[y \mapsto [2,2], i \mapsto [3,3]]
\end{array}$$

By grouping configurations for the same statement and computing the lub's of their property states we obtain the five configurations:

$$\begin{array}{l}
\langle y := 2; i := 0; \textbf{while } i \leq y \textbf{ do } i := i+1, ps\rangle \\
\langle i := 0; \textbf{while } i \leq y \textbf{ do } i := i+1, ps[y \mapsto [2,2]]\rangle \\
\langle \textbf{while } i \leq y \textbf{ do } i := i+1, ps[y \mapsto [2,2], i \mapsto [0,3]]\rangle \\
\langle i := i+1; \textbf{while } i \leq y \textbf{ do } i := i+1, ps[y \mapsto [2,2], i \mapsto [0,2]]\rangle \\
ps[y \mapsto [2,2], i \mapsto [3,3]]
\end{array}$$

From this we can infer that:

(a) At loop entry, variable y has the constant value 2. This allows the loop guard to be simplified to $i \leq 2$.

(b) In the loop, at the beginning of its body, the value of the array index i can be in the interval $[0, 2]$. So a reference to $x[i]$ at this point can give rise to an array–out–of–bounds error, since $x[2]$ is undefined.

---

3 Level A

For grade B you need to have passed level C and obtained 4 (out of 10) points from this section. For grade A you need 7 points from this section.

1. As we discussed in the course, statement *equivalence*: $\boxed{\text{5p}}$

$$S_1 \sim S_1'$$

can be used to justify formally program transformation and optimization: if program S_1' is considered "better" (in some meaningful sense) than program S_1, then the equivalence justifies the replacement of S_1 by S_1'. However, equivalence in itself does not say that such a replacement is justified in *any* program context (i.e., where S_1 is a sub-program of some program S).

(a) Prove formally, in a semantic style of your choice, that statement equivalence justifies replacement in any program context. Mathematically speaking, this amounts to showing that statement equivalence is a *congruence*, i.e., that $\sim$ is preserved under the formation rules of the language: if $S_1 \sim S_1'$ and $S_2 \sim S_2'$ then $S_1; S_2 \sim S_1'; S_2'$, and similarly for the other non–atomic rules.

(b) Assume that we have proved in Hoare logic that $\{P\}\, S_1\, \{Q\}$ holds. Does $S_1 \sim S_1'$ then entail $\{P\}\, S_1'\, \{Q\}$, or do we have to re-verify the optimized program? Justify formally your answer.

Solution: Yes, it does, since $S_1 \sim S_1'$ implies $\mathcal{S}_{ds}[\![S_1]\!] = \mathcal{S}_{ds}[\![S_1']\!]$, and then:

$$\begin{array}{lll} & \models_{\text{par}} \{P\}\, S_1\, \{Q\} & \\ \Leftrightarrow & \forall s, s' \in \textbf{State}.\ (s \models P \text{ and } \mathcal{S}_{ds}[\![S_1]\!]\,(s) = s' \text{ imply } s' \models Q) & \{\text{Def. } \models_{\text{par}}\} \\ \Leftrightarrow & \forall s, s' \in \textbf{State}.\ (s \models P \text{ and } \mathcal{S}_{ds}[\![S_1']\!]\,(s) = s' \text{ imply } s' \models Q) & \{\mathcal{S}_{ds}[\![S_1]\!] = \mathcal{S}_{ds}[\![S_1']\!]\} \\ \Leftrightarrow & \models_{\text{par}} \{P\}\, S_1'\, \{Q\} & \{\text{Def. } \models_{\text{par}}\} \end{array}$$

2. While program verification is about *checking* the validity of assertions at given program points, abstract interpretation can be seen as a technique for *generating* valid assertions at program points. $\boxed{\text{5p}}$

(a) Connect the formal frameworks of abstract interpretation (with abstract domain $\mathbf{P}$, abstraction function $\mathsf{abs}_Z : \mathbf{Z} \to \mathbf{P}$, etc.) and the one of logical assertions expressing state properties. You can take as an example the Detection of Signs analysis.

Hint: Focus on state properties mentioning one program variable only.

Solution: Given an abstract domain of properties $\mathbf{P}$ and abstraction function $\mathsf{abs}_Z : \mathbf{Z} \to \mathbf{P}$, every property $p \in \mathbf{P}$ denotes the set of concrete values:

$$\mathsf{abs}_Z^{-1}(p) = \{z \in Z \mid \mathsf{abs}_Z(z) = p\}$$

Let $\psi_p(z)$ be any logic formula with z as its only free variable (in other words, unary *predicate*) that is true exactly when $\mathsf{abs}_Z(z) = p$. In the Detection of Signs analysis for instance, we can define $\psi_{\mathrm{NEG}}(z)$, $\psi_{\mathrm{ZERO}}(z)$ and $\psi_{\mathrm{POS}}(z)$ as the logic formulas $z < 0$, $z = 0$ and $z > 0$. We can then use such formulas as assertions over program variables x, with the obvious meaning that $\psi_p(x)$ holds in state s whenever $\psi_p(s(x))$ holds, i.e., whenever $\mathsf{abs}_Z(s(x)) = \mathsf{abs}(s)(x) = p$. For example, $x < 0$ holds in state s whenever $\mathsf{abs}(s)(x) = \mathrm{NEG}$, which is exactly what we want in order to relate the two frameworks. Notice that we only need to define predicates for the *atomic* properties of the abstract domain $\mathbf{P}$, since *lub*'s are easily formed using disjunction; furthermore, these predicates can usually be found in the definition of the abstraction function $\mathsf{abs}_Z : \mathbf{Z} \to \mathbf{P}$.

(b) How can one use abstract interpretation to generate valid assertions at given program points? Show your idea on an example.

Solution: Let's execute statement S in the abstract domain of the Detection of Signs analysis, from the initial property state TOP. This results in the property state $\mathcal{DS}[\![S]\!](\mathrm{TOP})$. Let $\mathcal{DS}[\![S]\!](\mathrm{TOP})(x) = \mathrm{NEG}$. Then execution of S in the concrete domain from any initial state s, if it terminates, will do so in a state where $x < 0$ holds. In this way we have generated the assertion $x < 0$ for the program point following statement S in any program of the shape (or equivalent to) $S; S'$.

(c) How can one use abstract interpretation to generate *loop invariants*? Show your idea on an example.

Solution: Let **while** b **do** S be a loop in our **While** language, and consider again the Detection of Signs analysis. Let p be a property. If we have:

$$\mathcal{DS}[\![\textbf{if } b \textbf{ then } S \textbf{ else skip}]\!](ps)(x) = ps(x)$$

for any property state ps such that $ps(x) = p$, then the logical assertion $\psi_p(x)$ must be an invariant of the loop **while** b **do** S. For example, if $ps(x) = \mathrm{NEG}$:

$$\mathcal{DS}[\![\textbf{if } \neg(x = 0) \textbf{ then } x := x - 2 \textbf{ else skip}]\!](ps) = ps[x \mapsto \mathrm{NEG}]$$

and therefore $x < 0$ is an invariant of the loop **while** $\neg(x = 0)$ **do** $x := x - 2$. Similarly, $x = 0$ is also an invariant of this loop. This idea leads to a simple strategy for generating loop invariants.